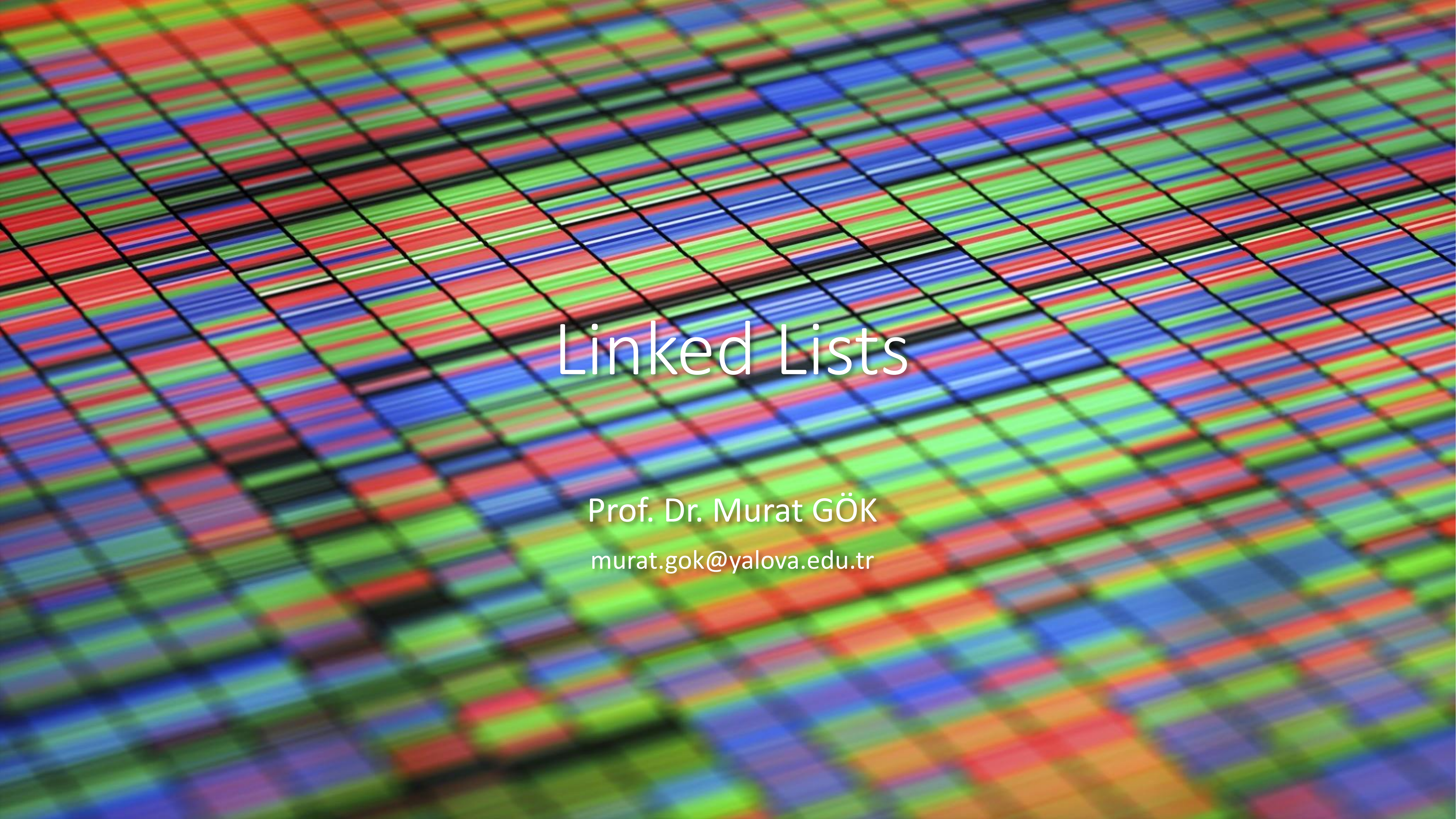


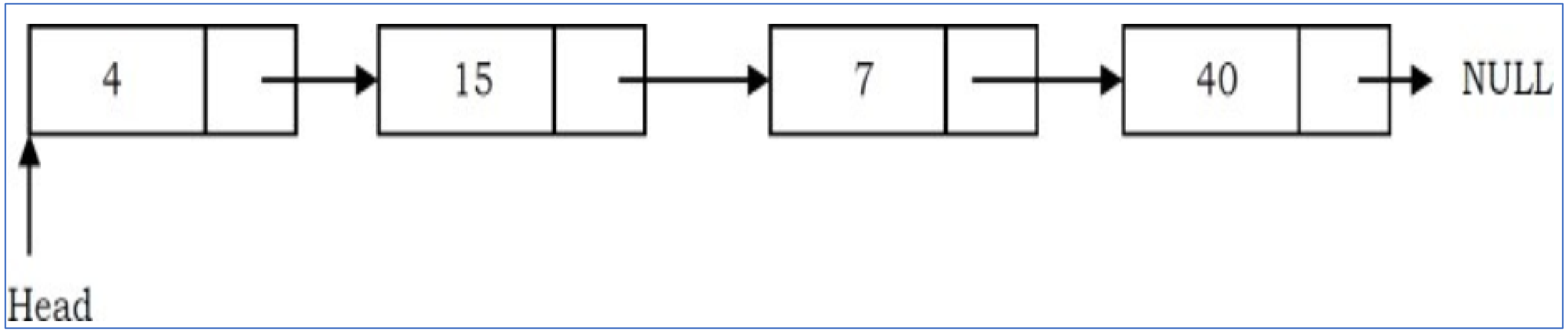
Linked Lists

Prof. Dr. Murat GÖK

murat.gok@yalova.edu.tr

Linked Lists

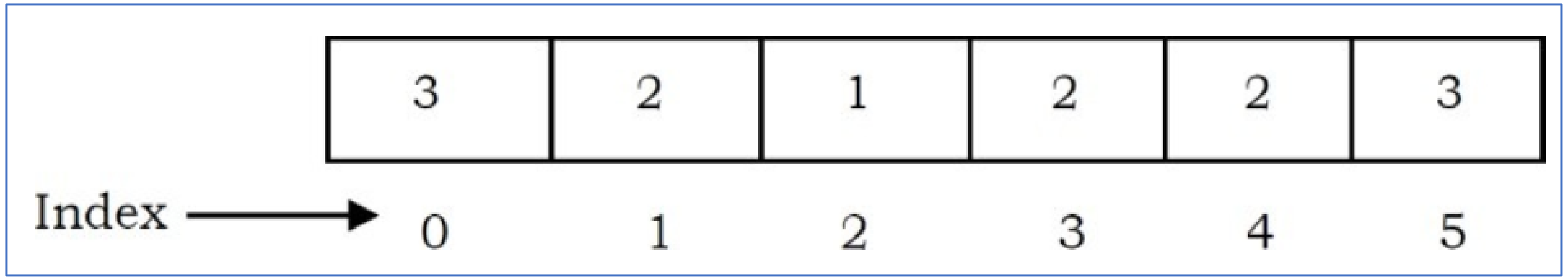
- A linked list is a data structure used for storing collections of data.



- A linked list has the following properties:
  - Successive elements (nodes) are connected by pointers.
  - The last element points to NULL.
  - Can grow or shrink in size during execution of a program.
  - Can be made just as long as required (until systems memory exhausts).
  - Does not waste memory space (but takes some extra memory for pointers). It allocates memory as list grows.

Arrays

- One memory block is allocated for the entire array to hold the elements of the array.
- The array elements can be accessed in constant time by using the index of the element as the subscript.



Linked Lists versus Arrays

1.Data Structure:

- **Array**: An array is a data structure that stores elements of the same data type in contiguous memory locations. Each element in an array is identified by its index.
- **Linked List**: A linked list is a data structure in which elements, called nodes, are connected together through pointers. Each node contains data and a reference to the next node in the list.

2.Insertion and Deletion:

- **Array**: Insertions and deletions in an array can be inefficient, especially if elements need to be inserted or removed from the middle or beginning of the array. It may require shifting elements to accommodate the change. This requires O(n) time.
- **Linked List**: Linked lists are more efficient for insertions and deletions, as you can simply update pointers to connect or disconnect nodes. Insertions and deletions in a linked list have a time complexity of O(1) for adding or removing elements at the beginning or end, and O(n) in the worst case for operations in the middle of the list.

Linked Lists versus Arrays

3. **Random Access:**
   - **Array**: Arrays provide fast random access to elements by their index. Accessing elements in an array has a time complexity of O(1).
   - **Linked List**: Linked lists do not support efficient random access. To access an element at a specific position, you need to traverse the list from the beginning, which has a time complexity of O(n).

4. **Memory Allocation:**
   - **Array**: Arrays have a fixed size, which is determined when they are created. If you need to change the size of an array, you may need to create a new array and copy elements from the old array.
   - Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.
   - **Linked List**: Linked lists can dynamically allocate memory for each node as needed, which makes them more flexible in terms of memory usage. They can grow or shrink as elements are added or removed.

Linked Lists versus Arrays

5. Memory Overhead:

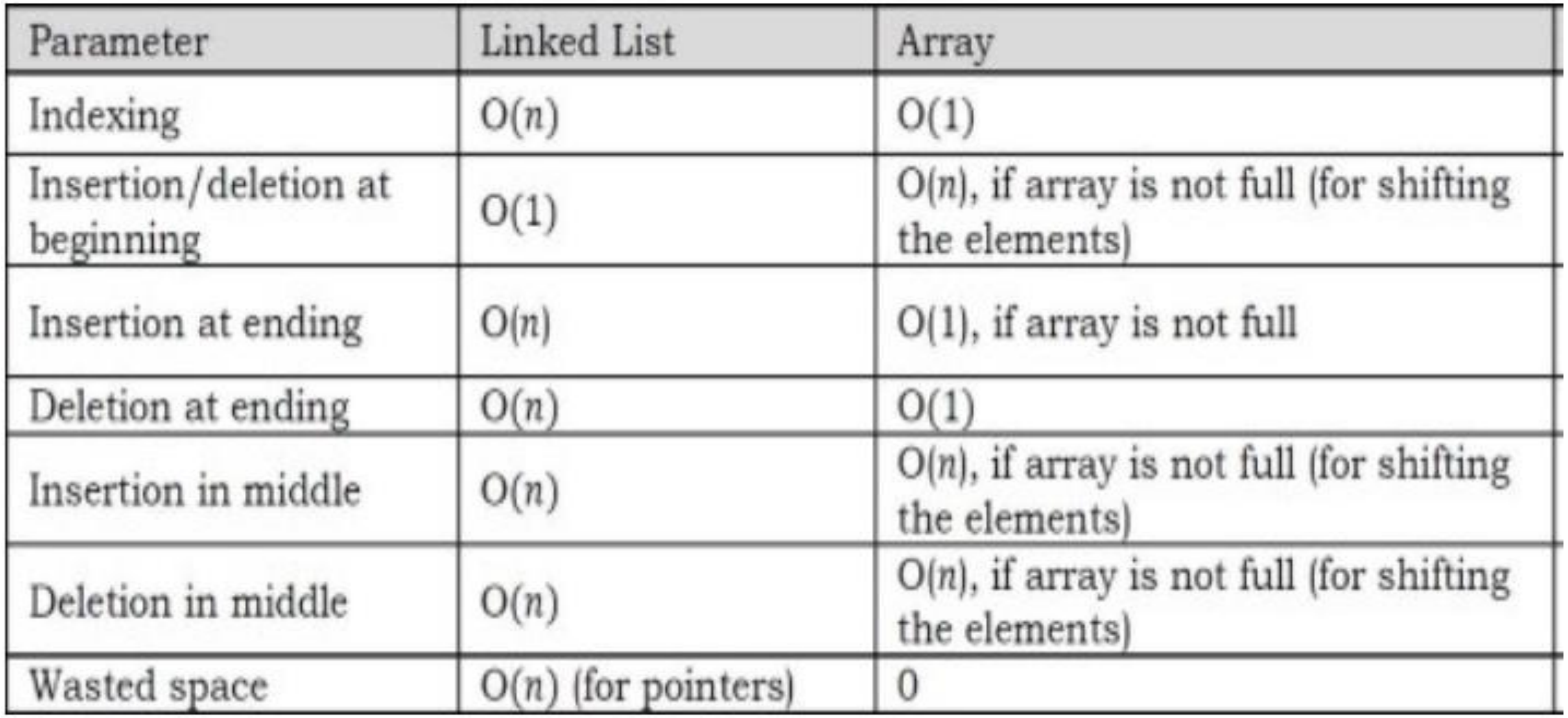
- **Array**: Arrays have less memory overhead compared to linked lists because they only need to store the data elements and their indices.
- **Linked List**: Linked lists have additional memory overhead for storing the pointers/references to the next node, making them less memory-efficient.

6. Implementation Complexity:

- **Array**: Arrays are simpler to implement and have better cache locality, which can lead to faster access times in practice.
- **Linked List**: Linked lists are more complex to implement and may result in worse cache performance due to scattered memory access patterns.

- **In summary**, the choice between linked lists and arrays depends on the specific requirements of your application. If you need fast random access and a fixed-size collection, arrays are a better choice. If you require efficient insertions and deletions, dynamic sizing, and can tolerate slower access times, linked lists may be more suitable.

Linked Lists versus Arrays



Parameter	Linked List	Array
Indexing	O(n)	O(1)
Insertion/deletion at beginning	O(1)	O(n), if array is not full (for shifting the elements)
Insertion at ending	O(n)	O(1), if array is not full
Deletion at ending	O(n)	O(1)
Insertion in middle	O(n)	O(n), if array is not full (for shifting the elements)
Deletion in middle	O(n)	O(n), if array is not full (for shifting the elements)
Wasted space	O(n) (for pointers)	0

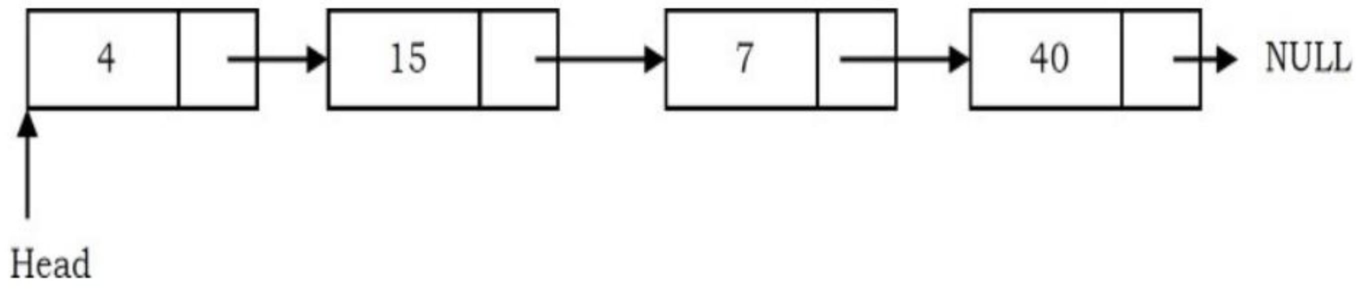
Linked List Types and Operations

- Linked List Types
  - Singly linked lists
  - Doubly linked lists
  - Circular lists

- Main Linked Lists Operations
  - Traversal: Traverses the list.
  - Insert: Inserts an element into the list.
  - Delete: Removes and returns the specified position element from the list.

Singly Linked Lists

- Generally, “linked list” means a singly linked list. This list consists of a number of nodes in which each node has a next pointer to the following element. The link of the last node in the list is NULL, which indicates the end of the list.



- **Traversing the Linked List:**
  1. Follow the pointers.
  2. Display the contents of the nodes (or count) as they are traversed.
  3. Stop when the next pointer points to NULL.

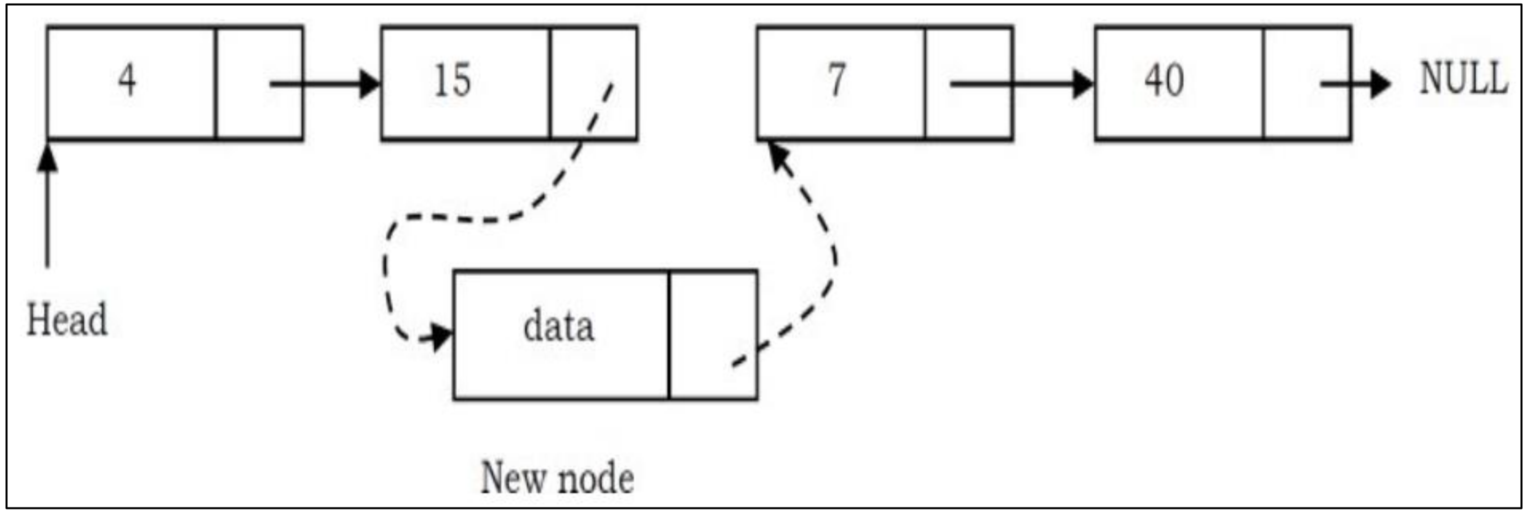
Traversing the Linked List

code

```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = ListNode(value)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")

# Create a linked list and add integers to it
linked_list = LinkedList()
linked_list.append(10)
linked_list.display()  # Display the list after adding 1
linked_list.append(20)
linked_list.display()  # Display the list after adding 2
linked_list.append(30)
linked_list.display()  # Display the list after adding 3
```

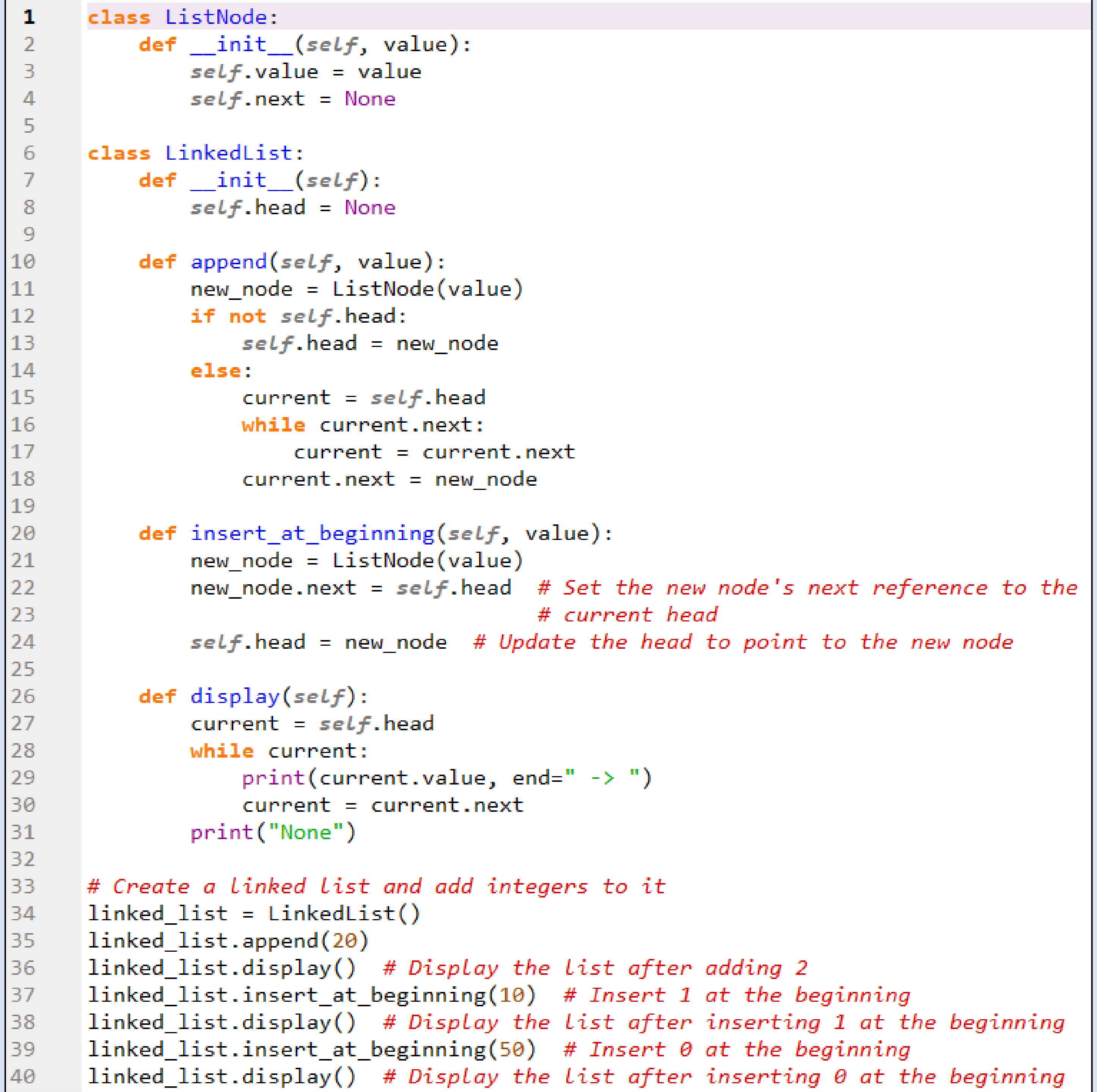
Singly Linked Lists

- **Singly Linked List Insertion:**
- Insertion into a singly-linked list has three cases:
  1. Inserting a new node before the head (at the beginning)
  2. Inserting a new node after the tail (at the end of the list)
  3. Inserting a new node at the middle of the list (random location)



Inserting A New Node Before The Head

code



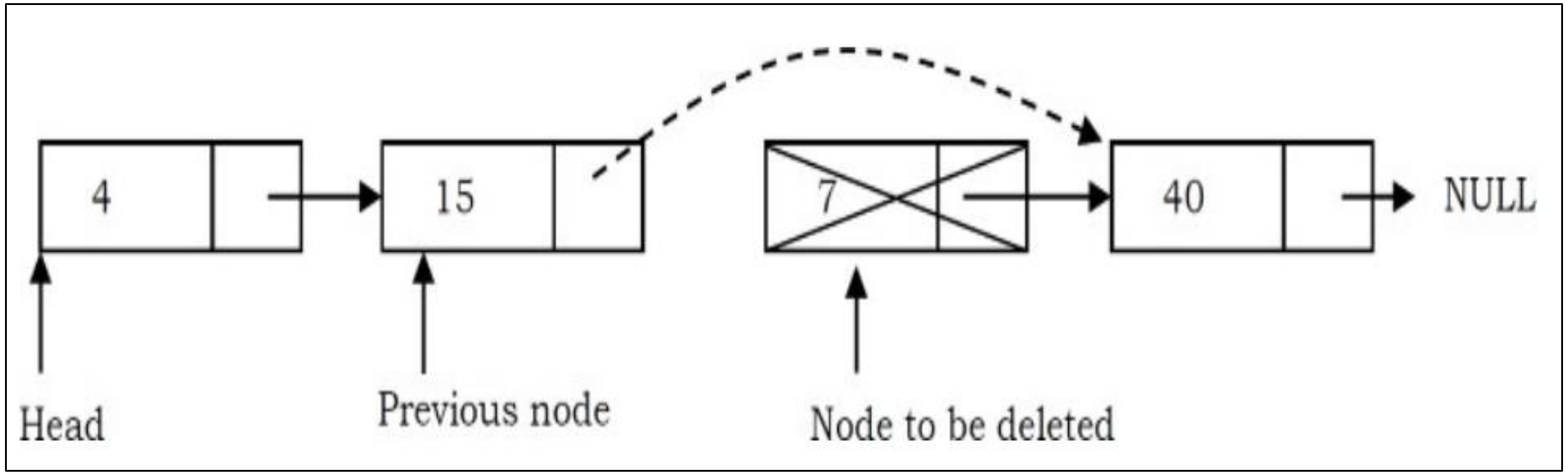
```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = ListNode(value)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def insert_at_beginning(self, value):
        new_node = ListNode(value)
        new_node.next = self.head  # Set the new node's next reference to the
                                   # current head
        self.head = new_node  # Update the head to point to the new node

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")

# Create a linked list and add integers to it
linked_list = LinkedList()
linked_list.append(20)
linked_list.display()  # Display the list after adding 2
linked_list.insert_at_beginning(10)  # Insert 1 at the beginning
linked_list.display()  # Display the list after inserting 1 at the beginning
linked_list.insert_at_beginning(50)  # Insert 0 at the beginning
linked_list.display()  # Display the list after inserting 0 at the beginning
```

Inserting A New Node Anywhere in The List

```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = ListNode(value)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def insert_at_middle(self, value, position):
        if position < 0:
            print("Invalid position")
            return

        new_node = ListNode(value)
        if position == 0:
            new_node.next = self.head
            self.head = new_node
            return

        current = self.head
        index = 0

        while current and index < position - 1:
            current = current.next
            index += 1

        if current is None:
            print("Position is out of range")
        else:
            new_node.next = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")

# Create a linked list and add integers to it
linked_list = LinkedList()
linked_list.append(20)
linked_list.display()  # Display the list after adding 2
linked_list.insert_at_middle(70, 0)  # Insert 1 at the beginning
linked_list.display()  # Display the list after inserting 1 at the beginning
linked_list.insert_at_middle(50, 2)  # Insert 3 in the middle
linked_list.display()  # Display the list after inserting 3 in the middle
```

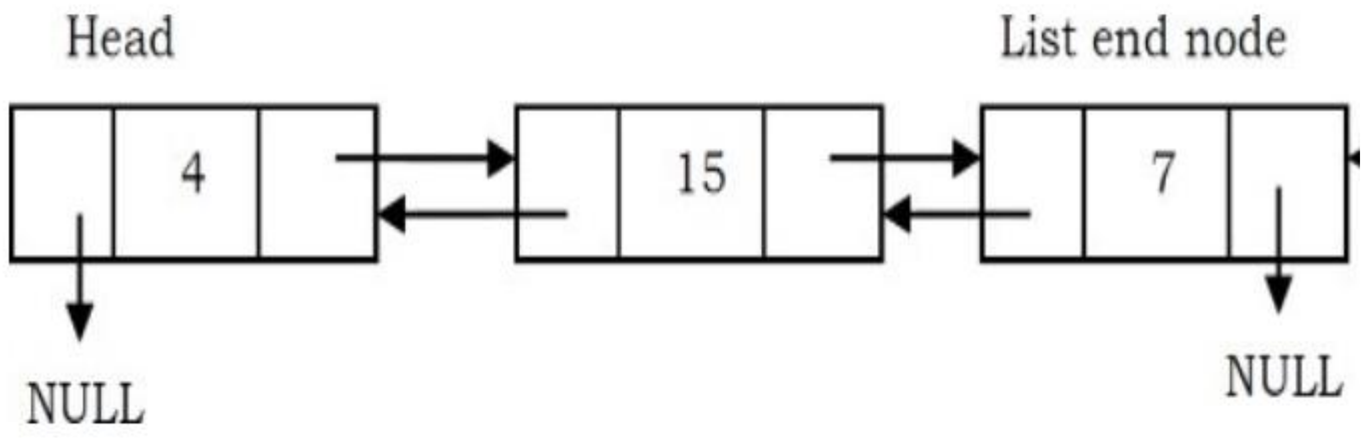
Singly Linked Lists

- **Singly Linked List Insertion:**
- Like insertion, here we also have three cases:
  1. Deleting the first node,
  2. Deleting the last node,
  3. Deleting an intermediate node.



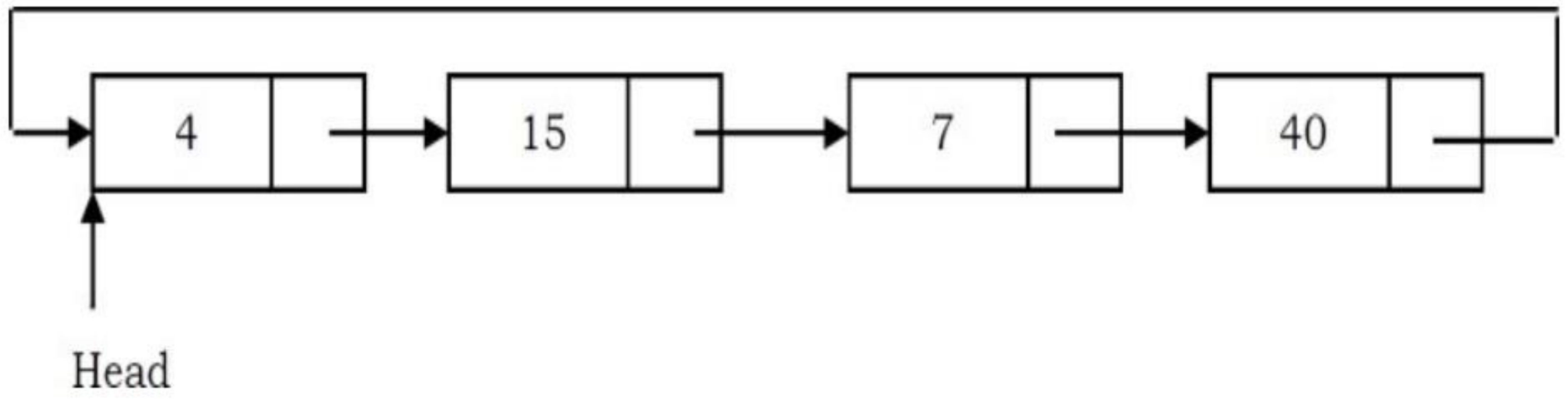
Doubly Linked Lists

- The advantage of a doubly linked list (also called two – way linked list) is that given a node in the list, we can navigate in both directions.
- Adding or deleting a node is made easier by eliminating the need to traverse to the previous node.
- The primary disadvantages of doubly linked lists are:
- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations).



Circular Linked Lists

- In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value. But circular linked lists do not have ends.
- While traversing the circular linked lists we should be careful; otherwise, we will be traversing the list infinitely.
- In circular linked lists, each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list.
- In some situations, circular linked lists are useful. For example, when several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes do (round robin algorithm).



Thank you for
your attention